Programming Techniques Using Character Sets and Character Mappings in Icon*

Ralph E. Griswold

TR 78-15a

December 6, 1978

Department of Computer Science

The University of Arizona

*This work was supported by the National Science Foundation under Grant MCS75-01307.

Programming Techniques Using Character Sets and Character Mappings in Icon*

Ralph E. Griswold

Department of Computer Science
The University of Arizona

1. Introduction

The character set and character mapping facilities in Icon, used in conjunction with its string-processing facilities, support a number of unusual programming techniques that can be used to advantage in a variety of nonnumerical programming problems.

This paper descibes the features that are important to these techniques and characterizes their usage. Examples are given to illustrate the major paradigms. A familiarity with Icon [1,2] is assumed. This paper uses Icon constructs freely, supplementing them with additional notation as required.

2. Character Sets

There are a variety of character sets in use on different kinds of computers. They differ in size, in the relationship between the internal representations of characters to control functions and external graphics, and (hence) in collating sequence. The most commonly used character sets are ASCII [3], EBCDIC [4], and various forms of BCD [5]. Internally, a character is simply an integer in the range from 0 to one less than the size of the character set. Thus in ASCII, there are 128 characters with internal representations from 0 to 127 (decimal), inclusive.

Most of the programming techniques described in this paper depend on the use of characters within a program, rather than their input or output. Where graphic representations are important, it is desirable, but not necessary, to have both upper- and lower-case letters but any of the common collating sequences will suffice. The size of the character set is significant, however, since in a number of applications individual characters are used to represent or label other objects.

The size of the internal Icon character set is 256. This character set is independent of the size of the character set for the host computer on which Icon runs. The internal character set and the host character set are interfaced only by input and output routines. Despite the size of its character set, Icon is ASCII based and the first 128 characters have ASCII interpretations. The use of the remaining characters is illustrated in subsequent examples. It is assumed for ease of presentation that both upper- and lower-case letters are available on the host machine. This assumption is not essential, however, since Icon

---

*This work was supported by the National Science Foundation under Grant MCS75-01307.

provides escape conventions for the literal representation of any internal character, regardless of input limitations that may be imposed by the host computer [2].

Icon supports a character set type, called cset for short. In Icon csets may have from 0 to 256 members. The value of the keyword &cset is a cset containing all 256 characters.

Csets are constructed from strings using the built-in function cset(s), which produces a cset consisting of the characters in the string s. While a string may contain duplicate characters, a cset cannot, of course. Similarly, the order of characters in s is irrelevant to the resulting cset. Thus

```
cset("armada")
cset("ramada")
cset("drama")
cset("dram")
```

all produce equivalent csets.

Historical Note: The concept of character set is latent in SNOBOL4 [6] and related languages. Although there is no character set type in SNOBOL4, typical implementations of SNOBOL4 deal with various representations of character sets [7,8] in order to support lexical analysis functions such as SPAN(S) and BREAK(S). The emergence of characters sets in Icon is a linguistic elevation of an implementation mechanism to full status as a source-language feature. The consequences of this elevation exceed the mechanisms for which they were originally developed, however.

Aside from type conversions, there are five built-in operations defined on csets:

~c	complement with respect to &cset
c_1 ++ c_2	union
c_1 ** c_2	intersection
c_1 -- c_2	difference

The creation of a cset from a string may be considered to be type conversion. Conversely, a cset may be converted to a string using the built-in function string(c). In this operation, the resulting string is alphabetized, that is, the characters of c are placed in the string according to their relative position in the collating sequence. For example,

```
alpha := string(&cset)
```

assigns to alpha a string consisting of all the available characters in order of their collating sequence. This string is referred to from place to place throughout this paper.

As a consequence of the properties of these conversions, the result of

s_2 := string(cset(s_1))

is a string s_2 which contains every distinct character of s_1 arranged in alphabetical order. This feature can be used to advantage, as is described in later sections.

Icon also supports implicit type conversions, coercing arguments to expected types as the context demands. For example, $s_1 \,||\, s_2$ is the concatenation of strings s_1 and s_2. Similarly, if c_1 and c_2 are csets, $c_1 \,||\, c_2$ produces a string that is the concatenation of the results of converting c_1 and c_2 to strings.

3. Character Mappings

Icon has an apparently innocuous built-in function for mapping the characters in a string, map(s_1,s_2,s_3). This function produces a result in which every character of s_1 that appears in s_2 is replaced by the corresponding character in s_3. For example, the result of

```
map("retroactive","aeiou","-----")
```

is "r-tr--ct-v-". Different characters can also be mapped differently. The result of

```
map("retroactive","aeiou","AEIOU")
```

is "rEtrOActIvE".

Note: The function map in Icon is virtually identical to the function REPLACE in SNOBOL4, the difference being in the handling of the case in which the lengths of s_2 and s_3 are different. In Icon, this is an error, while in SNOBOL4, it causes failure of the evaluation of the function.

3.1 Properties of Character Mappings

The description of the map function given above is a superficial one. In order to use the full capabilities of this function, it is necessary to be more precise about the operation and its consequences. In the discussion that follows, the form of the call is

s_4 := map(s_1,s_2,s_3)

1. The length of s_4 is the same as the length of s_1, regardless of the values of s_2 and s_3. In Icon terms, this is stated as

size(s_4) = size(s_1)

To remain in the domain of Icon as much as possible, this terminology is used subsequently.

If the notation $alpha^n$ is used for the set of all strings of length n that are composed of characters in alpha, then in general the result of the operation is a many-to-one mapping of $alpha^{size(s_1)}$ into itself.

2. The relative order of characters of s_2 and s_3 is significant,

since it establishes the correspondence used in the mapping. Thus the two expressions

```
map(s1,"aeiou","AEIUO")
map(s1,"uoiea","UOIEA")
```

produce the same result, but the two expressions

```
map(s1,"aeiou","AEIOU")                    (1)
map(s1,"uoiea","AEIOU")                    (2)
```

produce quite different results, in general.

As an aid to visualization, the correspondences between characters in s_2 and s_3 are shown as maps indicating the correspondences between individual characters directly. The map for expression (1) is

s_2	a	e	i	o	u
	↓	↓	↓	↓	↓
s_3	A	E	I	O	U

Expression (2) has the map

s_2	u	o	i	e	a
	↓	↓	↓	↓	↓
s_3	A	E	I	O	U

Note that only the relative order is important. Thus the map

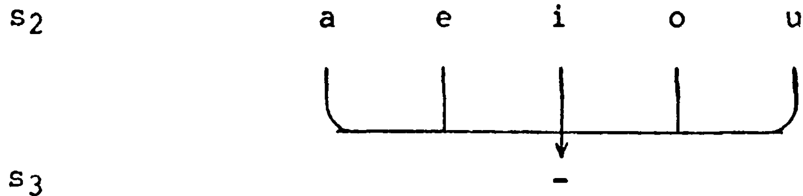
s_2	a	e	i	o	u
	↓	↓	↓	↓	↓
s_3	U	O	I	E	A

is equivalent to the previous map. The expression

```
map(s1,"aeiou","UOIEA")
```

is also equivalent to expression (2).

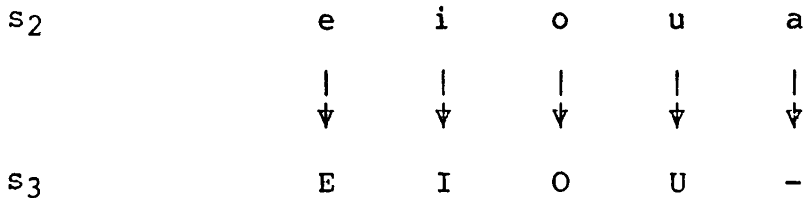
3. As illustrated in the first example in this section, s_3 may contain duplicate characters. This results in a mapping that is illustrated as follows



4. Duplicate characters in s_2 are permitted. In this case the last (rightmost) correspondence with s_3 holds. For example, the map for

map(s_1,"aeioua","AEIOU-")

is



It is convenient for the purposes of discussion to deal with the *reduced forms* of s_2 and s_3, in which there are no duplicate characters in s_2. In addition, it is convenient to deal with *canonical forms* in which s_2 is in reduced form and in alphabetical order and s3 is rearranged accordingly. The expression above in canonical form is

map(s_1,"aeiou","-EIOU")

The symbols $\hat{s}_2$ and $\hat{s}_3$ are used for the canonical forms of s_2 and s_3, respectively. See the end of Section 3.4 for a method of computing canonical forms.

In programming use, it is often convenient or more efficient to use values of s_2 and s_3 that are not canonical or even reduced. The map function can be thought of as performing the necessary canonicalization.

5. Characters of s_1 that do not occur in s_2 appear unchanged in their respective positions in s_4. The map function can be thought of as setting up automatic correspondences with such characters with themselves, but such detail is cumbersome and is omitted from maps shown in this paper. It is worth noting that

map(s_1,s_2,s_3)

and

map(s_1,alpha || s_2,alpha || s_3)

are equivalent.

6. s_1, s_2, and s_3 may be of any size, although the sizes of s_2 and s_3 must be the same, and size($\hat{s}_2$) = size($\hat{s}_3$) <= size(alpha). Furthermore, as noted above, size(s_4) = size(s_1).

3.2 Substitutions

The use of map(s_1,s_2,s_3) in which s_2 and s_3 are fixed and s_1 varies is called a substitution for s_1.

As a consequence of the properties listed in Section 3.1, the following condition holds:

Substitution Inverse Condition: For fixed s_2 and s_3 and varying s_1, the substitution

s_4 := map(s_1,s_2,s_3)

has an inverse if and only if $\hat{s}_3$ is equal to pi($\hat{s}_2$) for some permutation pi. An inverse is

s_1 := map(s_4,$\hat{s}_3$,$\hat{s}_2$)

The classical use for this kind of mapping occurs in cryptography. Substitution ciphers, which by definition must have inverses, are used to substitute for characters of a message. The form of substitution given above is directly applicable to monoliteral substitutions. See Reference 9 for an extended discussion and for programming techniques in SNOBOL4 that can be directly employed in Icon.

3.3. Permutations

The map function was originally designed to perform substitutions and its use for this purpose is obvious. It use to effect permutations (rearrangements) is less obvious.

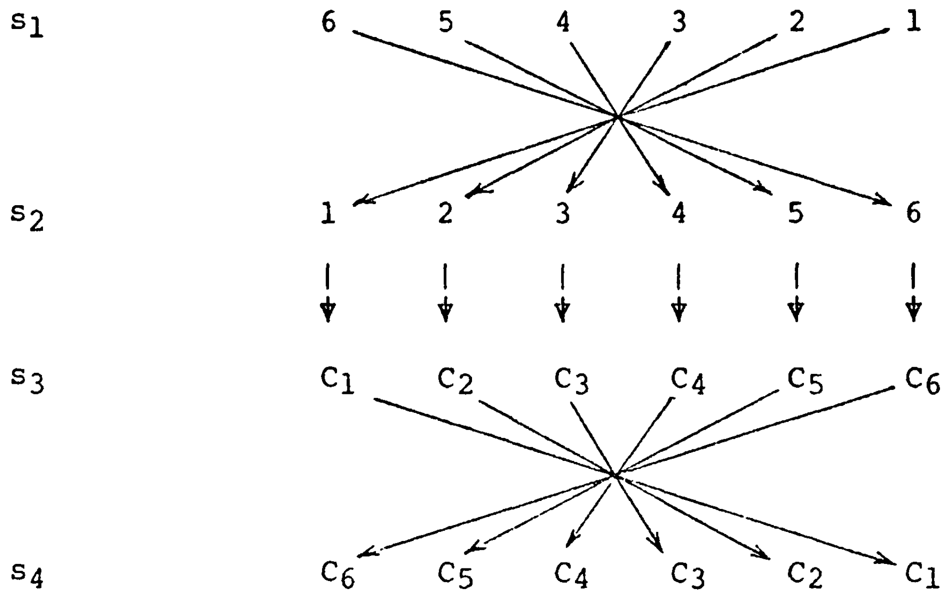
A simple example illustrates the technique. Suppose that the order of the characters of a string is to be reversed end-for-end. As a specific case, suppose size(s_3) = 6. Then

s_2 := "123456"
s_1 := "654321"
s_4 := map(s_1,s_2,s_3)

produces the desired result. In this expression, the mapping between s_2 and s_3 depends on the particular characters in s_3. If s_3 consists of characters $C_1C_2C_3C_4C_5C_6$, then the map is

s_2	1	2	3	4	5	6
	↓	↓	↓	↓	↓	↓
s_3	C_1	C_2	C_3	C_4	C_5	C_6

The desired permutation is accomplished since the characters of s_1 are mapped through s_2, by relative position, into those of s_3, as illustrated by the following diagram.



Historical Note: The use of character mapping to effect string reversal was first called to the author's attention in a private communication from Morris Seigel [10], who noted the technique is a use of the IBM 360 translate instruction [11]. This specific use was mentioned in the second edition of the SNOBOL4 programming language manual [6]. Jim Gimpel subsequently generalized the technique, which is described in Reference 12. A more extensive, but less formal presentation is given in Reference 9.

From the example above, it is clear that the technique can be used to perform any permutation, provided s_3 is not longer than size(alpha). Specifically:

Permutation Property: If pi is a permutation on a string of size n <= size(alpha) and s_2 is a string of n distinct characters, then the result of

s_4 := map(pi(s_2),s_2,s_3)

is s_4 = pi(s_3). Furthermore, an inverse to the permutation is

s_3 := map(s_2,pi(s_2),s_4)

Note that for constant values s_2 and pi(s_2), the first expression above applies the permutation pi to all strings s_3 of size n.

An application of fixed permutations applied to a set of strings occurs again in classical cryptography, where various transposition ciphers (route transposition, columnar transposition, and so forth) can all be seen as instances of this paradigm [9].

3.4. Positional Transformations

Permutations are a restricted case of more general positional transformations [12]. A positional transformation rho(s) of a string s is a rearrangement of the charcters of s in which

(1) Any character in a specific position in s may appear in zero or more fixed positions in rho(s).

(2) Additional constant characters, independent of the characters in s may appear in rho(s) at other fixed positions. These characters are called nulls.

For example, (abc)(cba) is a positional transformation of abc. The same positional transformation applied to xxy produces (xxy)(yxx). In this example, the parentheses are nulls.

Positional Transformation Property: If rho(s) is a positional transformation, then the result of

s_4 := map(rho(s_2),s_2,s_3)

is s_4 := rho(s_3).

Obviously not all positional transformations have inverses. For example

s_4 := map("fl","flllll1",s_3)

produces a two-character string consisting of the first and last characters of a seven-character string s_3.

One form of positional transformation that always has an inverse is the permutation, as described in Section 3.3. The class of positional transformations with inverses is more general, however.

Positional Transformation Inverse Property: Given a positional transformation rho, the mapping

s_4 := map(rho(s_2),s_2,s_3)

has an inverse if and only if

(1) All the characters in s_2 are distinct.

(2) All characters in s_2 appear at least once in rho(s_2).

If these conditions hold, the inverse is

s_3 := map(s_2,rho(s_2),s_4)

In the first place, if there is a duplicate character in s_2, only the last correspondence with s_3 will hold, and a character of s_3 will be deleted in the transformation and hence cannot be restored, in general, by any mapping.

Similarly, it is easy to see that if rho(s_2) does not contain some character in s_2, then the corresponding character in s_3 will not appear in s_4 and hence cannot be restored by any mapping. It remains to show that characters of s_2 can occur more than once in rho(s_2) and that nulls in rho(s_2) do not affect the inverse mapping.

Consider a positional transformation in which a character of s_2 is duplicated.

$$
\begin{array}{ll}
s_2: & C_1C_2 \ldots C_n \\
\text{rho}(s_2): & C_1C_2 \ldots C_nC_1 \\
s_3: & D_1D_2 \ldots D_n
\end{array}
$$

Then the map has the form

$$
\begin{array}{lcccc}
s_2 & C_1 & C_2 & \cdots & C_n \\
 & \downarrow & \downarrow & \cdots & \downarrow \\
s_3 & D_1 & D_2 & \cdots & D_n
\end{array}
$$

and s_4 is clearly $D_1D_2 \ldots D_nD_1$. When the inverse transformation is applied, rho(s_2) and s_4 stand in the correspondence

$$
\begin{array}{lccccc}
\text{rho}(s_2): & C_1 & C_2 & \cdots & C_n & C_1 \\
 & \downarrow & \downarrow & \cdots & \downarrow & \downarrow \\
s_4: & D_1 & D_2 & \cdots & D_n & D_1
\end{array}
$$

so the map for the reduced form is

$$
\begin{array}{cccc}
C_1 & C_2 & \cdots & C_n \\
\downarrow & \downarrow & \cdots & \downarrow \\
D_1 & D_2 & \cdots & D_n
\end{array}
$$

which is clearly an inverse to the original map. That is, duplicate characters of s_2 in rho(s_2) always stand in the same correspondence to characters of s_3. Furthermore, this applies to any rearrangement or duplication of C_1, C_2, ..., C_n in rho(s_2), since duplicate characters always produce identical correspondences.

Consider next the case in which the positional transformation contains a null X_1. For simplicity, suppose rho(s_2) has the form

$$
\text{rho}(s_2): \quad C_1C_2 \ldots C_nX_1
$$

Then s_4 will have the form

$$s_4: \qquad D_1D_2...D_nX_1$$

and in the inverse transformation the following correspondences hold:

$$\begin{array}{lccccc} rho(s_2): & C_1 & C_2 & \ldots & C_n & X_1 \\ & \downarrow & \downarrow & \vdots & \downarrow & \downarrow \\ s_4: & D_1 & D_2 & \ldots & D_n & X_1 \end{array}$$

Since by definition X_1 does not occur in s_2, it will not appear in the result.

It is easy to show that the same situation exists for other nulls and that their location in rho(s_2) is irrelevant.

Note: The canonical forms in the substitution paradigm can be obtained as follows:

```
ŝ2 := string(cset(s2))
ŝ3 := map(ŝ2,s2,s3)
```

This mapping is the inverse of the positional transformation that maps $\hat{s}_2$ into s_2.

Positional transformations with inverses appear in classical transposition ciphers, such as grilles [9], in which null characters are added to the cipher to obscure the transposed message. It is interesteing to note, as well, that message characters can be duplicated in the cipher without interfering with the inverse deciphering process.

4. Applications and Examples

As mentioned above, many of the models for substitution and positional transformation are found in classical enciphering techniques. While there are no longer many practical applications of classical enciphering techniques, there are a number of related applications that are of interest. The examples that follow illustrate techniques that many be useful in such cases.

For brevity, program solutions are stripped down to their essentials. Tests for the validity of arguments and so forth are deliberately omitted; these components easily can be added.

4.1 Substitutions

Example 1: Case Folding

One of the common uses for substitution is to establish equivalences between characters by mapping one set into another. For example, it is often convenient to consider upper- and lower-case letters to be equivalent. Instances of this situation arise in command processors that are insensitive to case. To simplify processing, therefore, the input is "folded" into a single case. The following procedure maps upper-case letters into lower-case ones using the Icon keywords for these values:

```
procedure fold(s)
  return map(s,&ucase,&lcase)
end
```

Example 2: Bit String Operations

Bit strings can be simulated by character strings composed of zeroes and ones. The logical negation operation is then simply

```
procedure not(b)
  return map(b,"01","10")
end
```

The logical operations of "or", "and", and "exclusive or" can be performed by adding bit strings as integers and making appropriate substitutions:

```
procedure or(b1,b2)
  return map(b1+b2,"2","1")
end

procedure and(b1,b2)
  return map(b1+b2,"12","01")
end

procedure exor(b1,b2)
  return map(b1+b2,"2","0")
end
```

Note: In general it is necessary to perform symbolic addition, since bits strings of any reasonable size are too large to represent as integers on most computers. Furthermore, bit strings are usually considered to be of fixed length with leading zeroes as necessary. Therefore the expression b1+b2 above should be replaced by sum(b1,b2), where sum is a procedure that handles these problems.

Example 3: Displaying Card Decks

A related application of substitution is illustrated by the problem of manipulating and displaying a deck of cards. Here a standard deck of playing cards can be represented by 52 distinct characters. Although any 52 distinct characters can be used, it is convenient to use the upper- and lower-case letters, since

their graphic representations facilitate program development and debugging. Therefore

```
        deck := deckimage := &ucase || &lcase
```

provides a "fresh" deck. The identifier deckimage is retained as a labeling of the cards, while deck may, for example, be shuffled. Since individual characters are used to represent the cards, shuffling can be done easily by character exchanges [13]:

```
        procedure shuffle(deck) local m
          every m := size(deck) to 2 by -1 do
            deck[random(m)] :=: deck[m]
          return deck
        end
```

In order to display a shuffled deck, it is necessary to determine the suit and denomination of each card. Again, this can be done by a substitution in which the first 13 characters of deckimage are mapped into the character C (for clubs), the second 13 into D (for diamonds), and so on. The third argument to map in this case is

```
        suits := repl("C",13) || repl("D",13) || repl("H",13)
                   || repl("S",13)
```

Similarly, the denominations can be identified by associating the first character of each 13-character group of deckimage with A (for ace), the second character in each group by 2, and so on. The third argument of map in this case is

```
        denoms := repl("A23456789TJQK",4)
```

A simple display of a deck of cards is then provided by the following procedure

```
        procedure display(deck)
          global deckimage,suits,denoms
          write(map(deck,deckimage,suits))
          write(map(deck,deckimage,denoms))
          return
        end
```

This procedure displays the deck with the suits on the first line and the denominations directly below. For example, if the shuffled deck begins with the 3 of clubs, the ace of hearts, and the 8 of spades, and so on, the display has the following form:

```
CHS ...
3A8 ...
```

A refinement to this display is given in Section 4.2.

Note that the technique used above is independent of the character set of the host computer on which Icon runs. Even if the host character set is BCD, the procedures above will work properly, since internally Icon supports a larger character set. Thus

is is not necessary to change deckimage if the host character set does not support lower-case letters. The interface between the internal character set only occurs when the (upper-case) results are written out.

Example 4: Masking Characters

In order to isolate characters of interest from those that are not of interest, it is useful to map all uninteresting characters into a single "null" that is not in the set of interest. The following procedure substitutes the character s3 for all characters in s1 that are not contained in s2.

```
procedure mask(s1,s2,s3)
  return map(s1,~s2,repl(s3,size(~s2)))
end
```

For example,

```
mask("Watch for spooks","aeiou","-")
```

produces -a-----o-----oo-- .

An alternate form of coding that uses duplicate characters rather than character-set complementation is

```
procedure mask(s1,s2,s3)
  return map(s1,alpha || s2,repl(s3,size(alpha)) || s2)
end
```

Here a correspondence between each character of alpha (the string of all characters) and s3 is first established and then the correspondences of characters in s2 with themselves are appended to override their correspondences with s3.

Example 5: Extracting and Displaying Suits

In card games like bridge, it is customary to sort hands into suits and to order the suits by denomination. All the cards in the same suit can be extracted by substituting some null for all characters that are not in the desired suit. Standard templates for the suits can be set up as follows:

```
blanker := repl(" ",13)
denom := substr(&lcase,13)
clubs := denom || repl(blanker,3)
diamonds := blanker || denom || repl(blanker,2)
hearts := repl(blanker,2) || denom || blanker
spades := repl(blanker,3) || denom
```

The mapping to get the clubs, for example, is

```
suit := map(hand,deckimage,clubs)
```

The identifier denom is used to associate the cards of each suit with the same denominations, regardless of suit. For example, the 2 of clubs and the 2 of hearts are both mapped into b. In

each case, all characters that do not correspond to a given suit are mapped into a blank. Note that it is essential to select a null that is not among the characters used to represent the cards.

If the suit above is converted to a cset and back to a string, the result is an (alphabetized) version of the suit with a single instance of the null. A further substitution can be performed to get the corect visual representation of each card:

```
map(cset(suit),denom,"AKQJT98765432")
```

If the hand contains the ace, queen, ten, and two of clubs, the result would be AQT2.

Note that the null used here is "invisible" in printed output, although is it actually the first character in the string produced above (for the ASCII collating sequence). It can be removed, if desired, by performing the following operation instead:

```
map(differ(suit," "),denom,"AKQJT98765432")
```

Note that in any case the final mapping to get the desired visual representation is done after the formation of the cset, since the visual representations are not in alphabetical order according to rank.

Other Applications

A number of other interesting uses of substitution are given in Reference 12. Two examples are the translation of Roman numeral to a higher "octave" in the conversion of Arabic numerals, and the use of ten's-complement arithmetic to effect symbolic subtraction by addition.

4.2 Positional Transformations

Example 6: Reversal

The reversal of the order of characters in a string, as described in Section 3.3, is not of interest in itself, since there is a built-in function in Icon for performing this operation. The solution of the problem, however, serves as a model for a number of other positional transformations.

The approach is to provide, by conventional means, general templates for the transformation. The second argument of map serves as a labeling for the third argument, while the first argument is the desired permutation. The terms image and object are used to refer to these two strings, respectively. For reverse, a possible image, object, and corresponding template size are

```
revimage := "abcdefghijklmnopqrstuvwxyz"
revobject := "zyxwvutsrqponmlkjihgfedcba"
revsize := size(revimage)
```

and the procedure is

```
procedure reverse(s)
  global revimage,revobjct,revsize
  if size(s) <= revsize then
    return map(
      section(revobjct,-size(s)),
      section(revimage,size(s)),
      s
      )
  else
    return reverse(section(s,revsize+1))
      || map(revobjct,revimage,substr(s,1,revsize))
end
```

If s is not longer than the image template, the reversal is done in one mapping. In this case, specific templates of the correct length are selected from the general ones. Note that the first part of revimage is used, while the last part of revobjct is used. If s is too long, it is divided into two portions. One portion is reversed by a recursive call, while the other is reversed using the full templates. This process can also be done iteratively at the expense of some complication of the code.

Note that the templates can be chosen in any convenient fashion, as long as revobjct is the reversal of revimage. For maximum efficiency in reversing long strings, the templates should be as long as possible: alpha and its reversal. These strings can be formed by conventional means:

```
revimiage := ""
every c := !alpha do
  revimage := c || revimage
```

In fact, these strings can be obtained by bootstrapping:

```
revimage := "ab"
revobjct := "ba"
revobjct := reverse(alpha)
revimage := alpha
```

This technique had the advantage of using the most elementary characterization of the positional transformation as well as avoiding possible errors in constructing the two long strings by conventional methods.

It is reasonable to question the use of map to effect this permutation, since it can be more easily coded by conventional techniques. One method is simply to concatenate successive characters in reverse order. The most compact Icon code for this method is

```
procedure reverse(s) local t
  every t := !s || t
  return t
end
```

Both this method and the mapping method are approximately time

linear in size(s) if secondary effects such as storage management anomalies are ignored. The conventional method is clearly linear. The map function itself is time linear in the sizes of its first and second arguments (see Section 6.2). In the procedure above, these two sizes are the same. Hence the mapping method is also time linear in size(s). Results of actual timings are shown in Fig. 1.

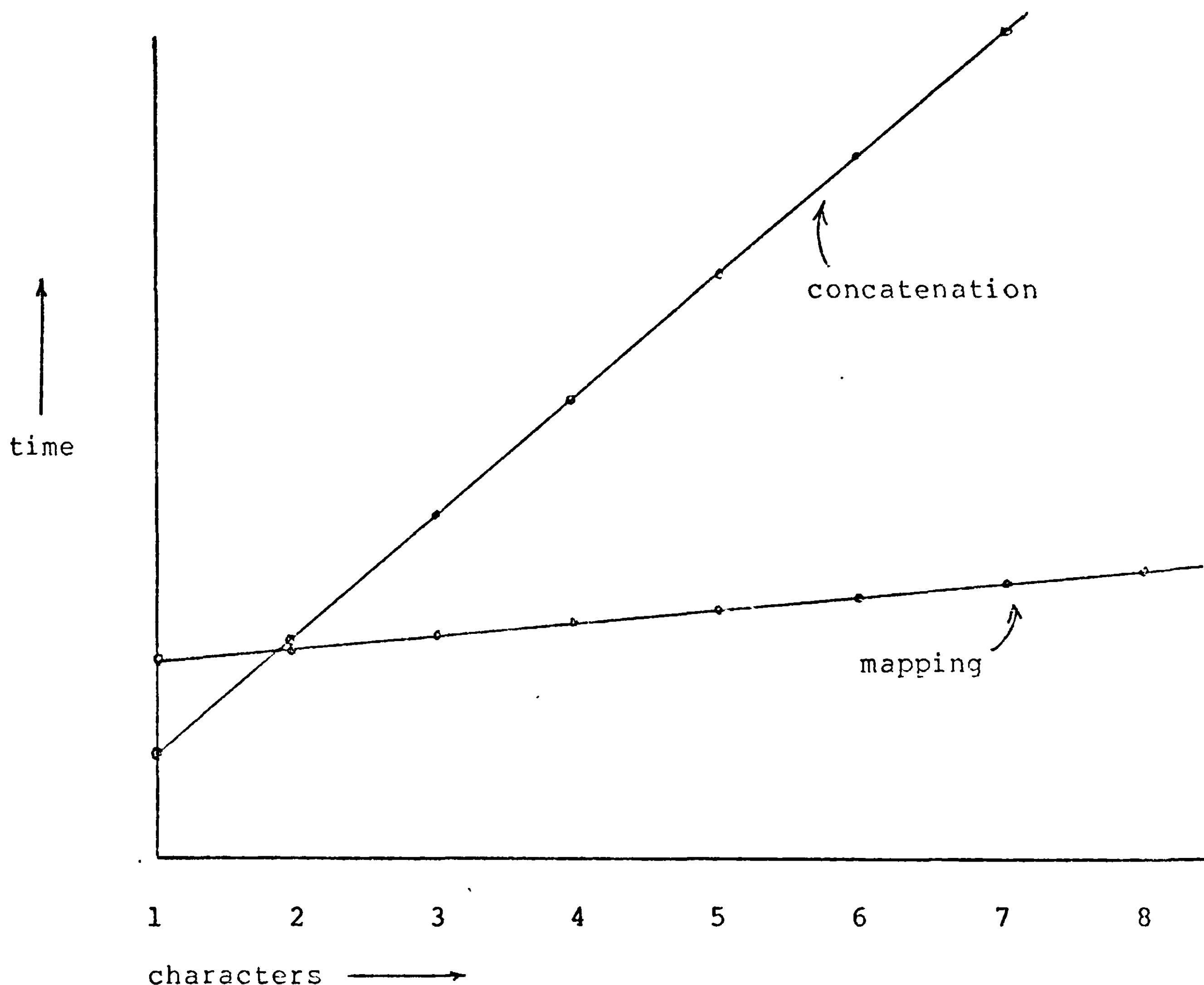


Fig. 1 -- Timings of String Reversal Methods

The interesting fact is that the (measured) constant of proportionality for the iterative method is nearly 8.5 times that of the mapping method. Furthermore, the cross-over point is at two characters! That is, the two methods take about the same amount of time for two-character strings, and the relative performance of the mapping method improves rapidly thereafter. Although the space requirements in terms of transient strorage are dependent on the details of internal storage management, the mapping method has the clear advantage of creating fewer intermediate strings. Part of the cost of transient allocation is shown in the relative constants of proportionality, but part is deferred in the form of garbage collection that may occur at unpredictable times to the

detriment of the conventional method. These remarks apply in general to the relative efficiency of effecting positional transformations by conventional means versus mapping.

Example 7: Character Exchange

A positional transformation that is similar to reversal is the exchange of adjacent characters in a string, For example, ABCDEF becomes BADCFE. The model for the solution to this problem is the same as reversal: an image to label the string to be transformed and an object that is the desired transformation of the labels. Suitable values are

```
ximage := "abcdefghijklmnopqrstuvwxyz"
xobjct := "badcfehgjilknmporqtsvexwzy"
xsize := size(ximage)
```

The procedure is virtually identical to the one for reversal, the difference being in the method for selection of the appropriate parts of the templates and the order of concatenation if the string is too long to be processed in one map:

```
procedure xchar(s)
  global ximage,xobjct,xsize
  if size(s) <= xsize then
    return map(
      substr(xobjct,1,size(s)),
      substr(ximage,1,size(s)),
      s
      )
  else
    return map(xobjct,ximage,substr(s,1,xsize))
      || xchar(section(s,xsize+1))
end
```

As with reversal, longer images and object provide more efficiency for processing long strings.

This example is included to illustrate an important aspect of this kind of positional transformation: the object must be a permutation of the image. In this case, this is only true if s is of even length. Suppose, for example, that the value of s is ABCDE. The map produced by the procedure above is

```
map("badcf","abcde","ABCDE")
```

Since the first argument contains a character, f, that does not appear in the second argument, this character is not changed by the mapping and appears in the result, which is BADCf, the last character being spurious. The procedure above only produces meaningful results for strings of even length. Of course, the exchange operation is not well defined for strings of odd length, which is the essential source of the problem. It is easy to add a check or modification to handle strings of odd length, but the problem is a general one and must be taken into account when performing positional transformations.

Example 8: Decollation

Both the positional transformations in the preceding examples are permutations. An example of a positional transformation that is not a permutation is decollation, the selection of every other character of a string. For example, to get the even characters, the following values can be used:

```
decimage := "-a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
decobjct := "abcdefghijklmnopqrst"
decsize := size(decimage)
```

with the procedure

```
procedure decollate(s)
  global decimage,decobjct,decsize
  if size(s) <= decsize then
    return map(
      substr(decobjct,1,size(s)/2),
      substr(decimage,1,size(s)),
      s
      )
  else
    return map(
      map(decobjct,decimage,substr(s,1,decsize))
        || decollate(section(s,decsize+1))
      )
end
```

Here only the even-numbered characters in the image have correspondences in the object and hence the result is the even-numbered characters in s. Any characters can be used as nulls in the image, provided that they are not the same as any of the labels for the even-numbered characters.

The odd-numbered characters can be selected by using the values above, but with a slightly modified image:

```
"a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

Since this value is just a one-character offset of the one above, the two operations can be combined into a single procedure decollate(s,n), where n is an integer whose parity, odd or even, determines whether the odd- or even-numbered characters are selected. A general-purpose procedure for decollation is

```
procedure decollate(s,n) local length
  length := size(s)
  n := mod(n,2)
  if length+n <= decsize then
    return map(
      substr(decobjct,1,(length+n)/2),
      substr(decimage,n+1,length),
      s
      )
  else
    map(
      substr(decobjct,1,(decsize-2)/2),
```

```
            substr(decimage,n+1,decsize-2),
            substr(s,1,decsize-2)
            )
              || decollate(section(s,decsize-1),n)
      end
```

The decollation of a smaller size than usual in the second section of the procedure allows for the fact that if n is odd, the substring of decimage starts at the second character. The choice of decsize-2 allows both parts of the decollation to operate on strings of even length, assuming s is of even length. An examination of this procedure will reveal that it operates correctly for strings of odd length. If s were split at decsize-1, however, the parity would have to be reversed for the second part.

Example 9: Collation

Strings can be collated as well as decollated by mapping. Since there are two strings specified in the collation process, it is useful to have two corresponding images, one to label each of the strings to be collated. The object is then the collation of these two images:

```
      colimage1 := "abcdefghijklm"
      colimage2 := "nopqrstuvwxyz"
      colobject := "anbocpdqerfsgthuivjwkxlymz"
      colsize := size(colimage1)
```

A collation procedure is

```
      procedure collate(s1,s2)
        global colimage1,colimage2,colobject,colsize
        if size(s) <= colsize then
          return map(
            substr(colobject,1,2*size(s)),
            substr(colimage1,1,size(s)) || substr(colimage2,1,size(s)),
            s1 || s2
            )
        else
          return map(
            colobject,
            colimage1 || colimage2,
            substr(s1,1,colsize) || substr(s2,1,colsize)
            )
              || collate(section(s1,colsize+1),section(s2,colsize+1))
      end
```

This procedure assumes that s1 and s2 are of the same length. It is instructive to examine the result when this condition is not satisfied.

Example 10: Displaying a Card Deck

The display of the deck of cards as done in Example 3 above produces as unattractive result. A more attractive display is obtained if the suit and denomination of each card are adjacent and there are separators (say blanks) between the representation of each card. Here there are 104 objects to be labeled (52 suit characters and 52 denomination characters) and some 156 characters in the result if one separating character is placed after the representation of each card. While the result can be obtained with a single map using long image and object strings, for display purposes it is more reasonable to divide the result into sections, say four sections of 13 cards each. For this purpose, convenient image and object strings are

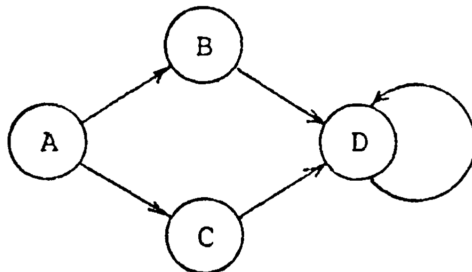
```
disimage := "ABCDEFGHIJKLMabcdefghijklm"
disobjct := "Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm"
```

where it is assumed that the upper-case letters label the suits and the lower-case letters label the denominations. The suit and denomination strings are then concatenated before mapping. A procedure is

```
procedure display(deck) local i
  global disimage,disobjct,deckimage,suits,denoms
  every i := 1 to 52 by 13 do
    write(
      map(
        disobjct,
        disimage,
        map(substr(deck,i,13),deckimage,suits)
        )
      )
end
```

Example 11: Directed Graphs

While it is customary to represent directed graphs by list structures or adjacency matrices, they can also be represented by character strings by associating a distinct character with each node and representing the arcs as character pairs. For example the graph



has arcs AB, AC, CD, BD, and DD. Represented as a single string, this graph is represented by

```
g := "ABACCDBDDD"
```

(If nodes without connecting arcs are allowed, a string containing all the nodes may be kept separately.) This representation is very compact and with the string processing operations of Icon, many graph operations can be performed economically. For example, a procedure to compute the number of nodes in a graph is simply

```
procedure nodecount(g)
  return size(cset(g))
end
```

Other graph operations are easily performed. For example, a cset of all nodes that are direct successors of other nodes is produced by

```
snodes := cset(decollate(g,2))
```

An example of the use of this representation is given by a procedure to determine the transitive closure of a node in a graph:

```
procedure closure(n,g)
  local  st,sn
  sn := n
  while (tn := sn ++ successors(sn,g)) ~=== sn do
    sn := tn
  return tn
end
```

The procedure successors(sn,g) returns all direct successors in g of nodes in the cset sn. Definition of this procedure is left as an exercise. The operation x ~=== y succeeds if x and y are different csets, so the loop continues until nothing new is added to the cset. It should be noted that all direct successors of the nodes in the evolving cset are added at each step.

Although the representation above is very compact and easy to manipulate, it is not suitable for display purposes. A positional transformation can produce a much more attractive result. Using an image and object of the form

```
grimage := "1234567890"
grobjct := "1 -> 2; 3 ->4; 5 -> 6; 7 -> 8; 9 -> 0;"
```

produces a display of the graph above as

```
A -> B; A ->C; C -> D; B -> D; D -> D;
```

It is a straightforward matter to generate longer image and object strings and to write a general-purpose procedure for producing the display.

Translation between various formats for input, output, display, and internal manipulation are easily derived in this manner.

Example 12: Biliteral Substitution

A classical ciphering technique is biliteral substitution, in which two characters are substituted for each character of the message. For example, DZ might be substituted for A, FR for B, and so on. This substitution is easily seen to be the collation of two simple substitutions, which can be performed as follows:

```
procedure bilit(s,image,first,second)
  return collate(
    map(s,image,first),
    map(s,image,second)
    )
end
```

where first and second are the two substitutions for the characters of image.

One use of this kind of "cipher" is in obtaining the hexadecimal representation of a character string. For ASCII, the values are

```
hex1 := repl("0",16) || repl("1",16) || repl("2",16) || repl("3",16)
  repl("4",16) || repl("5",16) || repl("6",16) || repl("7",16)
hex2 := repl("0123456789ABCDEF",8)
```

The keyword &ascii, consisting of a string of all ASCII characters in collating sequence, may be used for the image. For example, the value of

```
bilit("hello",&ascii,hex1,hex2)
```

is 68656C6C6F.

The use of this technique to convert character strings to their bit representations is left as an exercise.

Other Applications

References 9 and 12 provide numerous examples of positional transformations ranging from the reformatting of dates to the generation of pig latin.

5. Limitations

The main limitation on the programming techniques described in this paper are imposed by the limited size of the character set. In positional transformations, this is usually more of an annoyance than an actual limitation, since most positional transformations such as the reversal of a long string can be decomposed into a sequence of shorter transpositions. However, if the scope of the transposition requires more labels than there are characters in the character set, a different technique has to be used.

The really serious limitation occurs in the use of characters to represent distinct objects. The representation of a deck of playing cards in this way works nicely with any commonly used

character set, but that is merely a convenient coincidence. In the case of graphs, the representation used clearly limits the cases that can be handled. Furthermore, since the methods specifically rely on character operations, there is no way to extend the techniques if the size of the character set is inadequate.

6. Implementation

The techniques used to implement string and cset operations are only of interest here to the extent that they affect the efficiency of the programming techniques that have been described. See Reference 14 for a description of dynamic storage management in Icon and the details of data layout.

6.1 Character Sets

Character sets are represented as bit strings, with the bit in the position of the character in collating sequence set to 1 if the character is in the character set and set to 0 otherwise. The amount of space required for a cset depends on the size of the character set (256 in Icon), not on the number of characters it contains. In any event, csets require comparatively little storage space.

The construction of a cset from a string involves processing the characters of the string in sequence, setting the corresponding bit in the cset. This process is time linear in the size of the string.

Constructing a string from a cset involves the converse process and is also time linear in the number of characters in the cset.

Complementing a character set is time linear in the number of characters not in the character set, but is a comparatively fast operation compared to those that involve accessing characters. The other built-in character set operations are also time linear and correspondingly efficient.

6.2 Mapping

map(s_1,s_2,s_3) is performed by first building a table of correspondences between the characters of s_2 and those of s_3. This table contains one entry for each character in the character set (256 in Icon) and it is initialized by having each character correspond to itself. Then the entry for each character is s_2 is replaced by the corresponding character in s_3, working from left to right. Thus if there are duplicate characters in s_2, the last (right-most) correspondence results naturally.

Once the table is built, the characters in s_1 are processed in sequence and the result is built from the characters obtained by the entry in the table that corresponds to the character of s_1.

The amount of time required to build the table of correspondences is proportional to the size of s_2 and the amount of time required to do the actual mapping is proportional to the size of

s_1. Thus the total time required for the mapping is approximately

$$a*size(s_1) + b*size(s_2) + c$$

where c is constant overhead including the initialization of the table of correspondences.

The table of correspondences is static. The only storage allocation required for mapping is for the resulting string. Furthermore, if map is called successively with the same values of s_2 and s_3, the previous table of correspondences is used without reinitialization.

7. Conclusions

The character set and string processing facilities of Icon make programming techniques feasible that otherwise would require data to be represented in different ways. The main advantages of these techniques are the compactness of the data representations and the comparative efficiency of the operations.

This efficiency is largely obtained by the internalization of processes that would ordinarily involve loops at the source-language level. Specific examples of this are identifying distinct characters, sorting them using cset(s), and the positional transformations of long strings using a single mapping operation. Given appropriate computer architecture, character sets can be manipulated as bit vectors, with the potential improvement in efficiency that can be obtained from parallel operations [15].

It is interesting to note that csets are so useful in their role as sets independent of their relationships to specific characters, despite the limitation on the number of objects that can be represented. At the same time, csets provide an economical facility, largely because they _are_ limited in number.

There is no inherent reason why a language character set should be resticted to the character set of the host machine. Indeed, in the CYBER 175 implementation of Icon, the language character set is four times the size of the (standard) host character set and on the DEC-10 it is twice the size of the host character set. Character sets larger than those normally supported by any computer could easily be implemented, increasing the scope of the string processing facilities.

The problem of supporting a language character set that is different from the host character set is not as difficult as it might appear. In Icon, the size of a character (and hence of character sets) is an implementation parameter. Icon was originally configured for 128 characters and later changed to 256 characters to allow more flexibility. The change was easy and accomplished quickly. Furthermore, an internal character set that is independent of the host character set is an advantage, especially for enhancing portability, since the bulk of the system is written in machine-independent form with known collating sequence (ASCII is used). For example, the lexical analyzer is

machine independent, whereas if the internal character set varied according to the host character set of the target computer, there would be many complications.

The penalty for a larger character set is primarily in the space required for representing csets and strings. Doubling the size of the character set approximately doubles the amount of space required for storing a cset proper, although there is storage overhead that is independent of the size of the character set. Similarly, the larger the character set, the more space is required for each character of every string. The time for some operations is increased also. The larger size of strings may require more time in data movement and the number of items that have to be processed is increased for cset operations and the correspondences established in map.

Such "super character sets" would extend the domain of applicability of the techniques described in this paper. Although it is beyond to scope of this paper, a potentially more important advantage of very large character sets lies in their capacity to provide internal representations for larger sets of graphics than are supported by the host character set and hence in the processing of data for devices like phototypesetters. There are thorny problems related to the imbedding of the host character set in the character set of the language, especially with respect to "differences of opinion" about collating sequences.

Acknowledgement

I am indebted to Jim Gimpel for introducing me to character mappings. Students in my string and list processing classes have served in an exemplary manner as guinea pigs. In addition, David R. Hanson and John T. Korb have provided helpful suggestions on the presentation of the material in this paper.

References

1. Griswold, Ralph E., David R. Hanson, and John T. Korb. The Icon Programming Language; An Overview. Technical Report TR 78-3b, Department of Computer Science, The University of Arizona, Tucson, Arizona. October 2, 1978.

2. Griswold, Ralph E. User's Manual for the Icon Programming Language. Technical Report TR 78-14, Department of Computer Science, The University of Arizona, Tucson, Arizona. October 6, 1978.

3. American National Standards Institute. USA Standard Code for Information Interchange, X3.4-1968. New York, New York. 1968.

4. IBM Corporation. System/370 Reference Summary. Form GX20-1850-3. White Plains, New York. 1976.

5. Control Data Corporation. SCOPE Reference Manual. Publication Number 60307200. Sunnyvale, California. 1971.

6. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky.

The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

7. Griswold, Ralph E. The Macro Implementation of SNOBOL4; A Case Study in Machine-Independent Software Development. W. H. Freeman, San Francisco. 1972.

8. Gimpel, James F. "The Minimization of Spatially Multiplexed Character Sets", Communications of the ACM, Vol. 17, No. 6 (June, 1974). pp. 315-318.

9. Griswold, Ralph E. String and List Processing in SNOBOL4, Techniques and Applications. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1975.

10. Seigel, Morris M. Letter to author, October 12, 1969.

11. Computer Usage Company, Programming the IBM/360. John Wiley & Sons, New York. 1966. p. 208f.

12. Gimpel, James F. Algorithms in SNOBOL4. John Wiley & Sons, New York. 1976. pp. 46-51.

13. Knuth, Donald E. The Art of Computer Programming, Vol. 2. Addison-Wesley, Reading, Massachusetts. 1969. p. 125.

14. David R. Hanson. A Portable Storage Management System for the Icon Programming Language. Technical Report TR 78-16, Department of Computer Science, The University of Arizona, Tucson, Arizona. October 8, 1978.

15. Aho, Alfred V., John E. Hopcroft, and Jeffry D. Ullman. The Design and Analysis of Computer Algorithms. Addison Wesley Publishing Company, Reading, Massachusetts. 1976.